Modular Termination Verification: Extended Version

Bart Jacobs *Dragan Bosnacki* *Ruurd Kuiper*

Report CW 680, January 2015

KU Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Modular Termination Verification: Extended Version

Bart Jacobs *Dragan Bosnacki* *Ruurd Kuiper*

Report CW 680, January 2015

Department of Computer Science, KU Leuven

Abstract

We propose an approach for the modular specification and verification of total correctness properties of object-oriented programs. We start from an existing program logic for partial correctness based on separation logic and abstract predicate families. We extend it with *call permissions* qualified by an arbitrary ordinal number, and we define a specification style that properly hides implementation details, based on the ideas of using methods and bags of methods as ordinals, and exposing the bag of methods reachable from an object as an abstract predicate argument. These enable each method to abstractly request permission to call all methods reachable by it any finite number of times, and to delegate similar permissions to its callees. We illustrate the approach with several examples.

We then extend the approach to a concurrent setting, by incorporating an existing approach for verifying deadlock-freedom of channels and locks. Our main contribution here is to achieve information hiding by using method bags for lock ordering. We also show how our approach prevents infinite thread creation and enables verification of termination of fine-grained concurrent algorithms involving compare-and-swap loops.

We explain how our approach can be used also to verify liveness properties of non-terminating programs.

Modular Termination Verification: Extended Version

Bart Jacobs[1], Dragan Bosnacki[2], and Ruurd Kuiper[2]

[1] iMinds-DistriNet, Department of Computer Science, Leuven, Belgium
bart.jacobs@cs.kuleuven.be
[2] Eindhoven University of Technology, the Netherlands
{d.bosnacki,r.kuiper}@tue.nl

Abstract. We propose an approach for the modular specification and verification of total correctness properties of object-oriented programs. We start from an existing program logic for partial correctness based on separation logic and abstract predicate families. We extend it with *call permissions* qualified by an arbitrary ordinal number, and we define a specification style that properly hides implementation details, based on the ideas of using methods and bags of methods as ordinals, and exposing the bag of methods reachable from an object as an abstract predicate argument. These enable each method to abstractly request permission to call all methods reachable by it any finite number of times, and to delegate similar permissions to its callees. We illustrate the approach with several examples.
We then extend the approach to a concurrent setting, by incorporating an existing approach for verifying deadlock-freedom of channels and locks. Our main contribution here is to achieve information hiding by using method bags for lock ordering. We also show how our approach prevents infinite thread creation and enables verification of termination of fine-grained concurrent algorithms involving compare-and-swap loops.
We explain how our approach can be used also to verify liveness properties of non-terminating programs.

1 Introduction

We present our approach in the context of a simple Java-like programming language.

What should be the contract for interface IntFunc, such that we can prove total correctness properties of programs involving classes that implement the interface and classes that call the interface?

```
interface IntFunc {
  int apply(int x);
}
```

The contract should support programs such as shown in Figure 1.

We are not aware of existing work that answers this question.

```
class PlusN(int y) implements IntFunc {
  int apply(int x)
  { int y := this.y; x + y }
  static IntFunc createPlusN(int y)
  { new PlusN(y) }
}
class Twice(IntFunc f)
    implements IntFunc {
  int apply(int x) {
    IntFunc f := this.f;
    int y := f.apply(x);
    f.apply(y)
  }
  static IntFunc createTwice(IntFunc f)
  { new Twice(f) }
}
```

```
class Program {
  static void main() {
    IntFunc f1 :=
      PlusN.createPlusN(10);
    IntFunc f2 :=
      Twice.createTwice(f1);
    IntFunc f3 :=
      Twice.createTwice(f2);
    f3.apply(42)
  }
}
```

Fig. 1. A program that implements and calls interface IntFunc

This paper is structured as follows. In Sec. 2, we review the partial correctness approach that we start from. In Sec. 3, we introduce our approach for total correctness verification of sequential programs. In Sec. 4, we extend our approach to multithreaded programs. In Sec. 5, we show how to verify liveness of non-terminating programs. In Sec. 6, we briefly discuss our implementation. We discuss related work in Sec. 7 and we conclude in Sec. 8.

2 Separation logic and abstract predicate families

We start from an existing approach for modular specification and verification of partial correctness properties of object-oriented programs, based on separation logic [1, 2] and abstract predicate families [3]. A partial correctness specification for interface IntFunc is:

```
interface IntFunc {
  predicate IntFunc();
  int apply(int x);
    req this.IntFunc();
    ens this.IntFunc();
}
```

(Notice that we do not attempt to specify full functional correctness.) Interface IntFunc declares an *abstract predicate family* IntFunc with no parameters. Classes that implement the interface will instantiate this predicate family to denote access permissions for, and invariants over, the memory locations accessed by method apply. Annotations for the example program are shown in Figure 2.

```
class PlusN(int y) implements IntFunc {
  predicate IntFunc() = this.y ↦ _;
  int apply(int x)
    req this.IntFunc();
    ens this.IntFunc();
  { int y := this.y; x + y }
  static IntFunc createPlusN(int y)
    req true; ens result.IntFunc();
  { new PlusN(y) }
}
class Program {
  static void main()
    req true; ens true;
  {
    IntFunc f1 := PlusN.createPlusN(10);
    IntFunc f2 := Twice.createTwice(f1);
    IntFunc f3 := Twice.createTwice(f2);
    f3.apply(42)
  }
}
```

```
class Twice(IntFunc f)
    implements IntFunc {
  predicate IntFunc() =
    this.f ↦ f ∗ f.IntFunc();
  int apply(int x)
    req this.IntFunc();
    ens this.IntFunc();
  {
    IntFunc f := this.f;
    int y := f.apply(x);
    f.apply(y)
  }
  static IntFunc createTwice(IntFunc f)
    req f.IntFunc();
    ens result.IntFunc();
  { new Twice(f) }
}
```

Fig. 2. Example program annotated with partial correctness specifications

In the remainder of this section, we formally define the program logic that we start from.

The syntax of the programming language and the annotations is shown in Figure 3.

We assume a set of interface names $\iota \in \mathit{ItfNames}$ and a set of class names $C \in \mathit{ClassNames}$. The types τ of the programming language include at least the types **int** of integers and **bool** of booleans, and the interface types ι and class types C. Correspondingly, the values $v \in \mathit{Values}$ of the programming language include at least the integers $z \in \mathbb{Z}$ and the booleans $b \in \mathbb{B}$, and the object references $o \in \mathit{ObjRefs}$. We will assume additional types and values whenever useful for particular examples.

The expressions include the literal values v, the variable references x, and the pure operations $op(\overline{e})$ that map a sequence of argument values to a result value. The separation logic assertions P include the boolean expressions e, asserting that the expression evaluates to true; the points-to assertions $e.f \mapsto e$, asserting that the indicated field is present in the heap and holds the indicated value; the separating conjunction $P * P$, asserting that the heap can be split into two parts such that one conjunct holds in one part, and the other conjunct holds in the other part; regular conjunction and disjunction; and *predicate assertions* $e.p(\overline{e})$, asserting that the indicated predicate holds with the indicated argument values. p ranges over predicate names.

Like expressions, commands c return a value; unlike expressions, they may also access the heap and have side-effects. The commands include the expressions; a **let**-like construct $\tau\ x := c; c'$ that first executes c, binds the result to variable x of type τ, and then executes c'; conditional commands; parenthesized commands; static and instance method calls; object creation, field lookup, and field mutation commands.

A predicate declaration specifies a predicate name and a parameter list; a predicate definition additionally specifies a body.

An interface method specifies a return type, a method name, a parameter list, and a contract consisting of a precondition and a postcondition. A class method additionally specifies a kind (kind **instance** is the default and is usually left implicit) and a body.

An interface definition declares a number of predicate families and a number of interface methods. A class definition declares a list of fields (empty if omitted), an implemented interface (interface Empty, that declares no predicate families and no methods, if omitted), a number of predicate family instances, and a number of class methods.

The type definitions are the interface definitions and the class definitions. A program is a sequence of type definitions.

$$
\begin{aligned}
\tau &::= \mathbf{int} \mid \mathbf{bool} \mid \iota \mid C \mid \cdots \\
e &::= v \mid x \mid op(\overline{e}) \\
P &::= e \mid e.f \mapsto e \mid P * P \mid P \wedge P \mid P \vee P \mid e.p(\overline{e}) \\
c &::= e \mid \tau\ x := c; c \mid \mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else}\ c \mid \{\ c\ \} \\
&\quad \mid C.m(\overline{e}) \mid e.m(\overline{e}) \mid \mathbf{new}\ C(\overline{e}) \mid e.f \mid e.f := e \\
\mathit{pdecl} &::= \mathbf{predicate}\ p(\overline{\tau\ x}); \\
\mathit{pdef} &::= \mathbf{predicate}\ p(\overline{\tau\ x}) = P; \\
\mathit{imdef} &::= \tau\ m(\overline{\tau\ x});\ \mathbf{req}\ P;\ \mathbf{ens}\ P; \\
\mathit{mkind} &::= \mathbf{static} \mid \mathbf{instance} \\
\mathit{cmdef} &::= \mathit{mkind}\ \tau\ m(\overline{\tau\ x})\ \mathbf{req}\ P;\ \mathbf{ens}\ P;\ \{\ c\ \} \\
\mathit{idef} &::= \mathbf{interface}\ \iota\ \{\ \overline{\mathit{pdecl}}\ \ \overline{\mathit{imdef}}\ \} \\
\mathit{cdef} &::= \mathbf{class}\ C(\overline{\tau\ f})\ \mathbf{implements}\ \iota\ \{\ \overline{\mathit{pdef}}\ \ \overline{\mathit{cmdef}}\ \} \\
\mathit{tdef} &::= \mathit{idef} \mid \mathit{cdef} \\
\mathit{program} &::= \overline{\mathit{tdef}}
\end{aligned}
$$

Fig. 3. Syntax of the programming language and the annotations

We assume a function $\mathsf{classOf} : \mathit{ObjRefs} \to \mathit{ClassNames}$ such that infinitely many object references map to any given class. A heap $h \in \mathit{Heaps} = \mathit{ObjRefs} \times \mathit{FieldNames} \rightharpoonup \mathit{Values}$ is a partial function from pairs of object references and field names to values. We do not allow instantiation of classes that have no fields; therefore, the set of allocated objects can be derived from $\mathrm{dom}(h)$.

We define the semantics of programs by means of a big-step relation $(h, c) \Downarrow \gamma$ that relates a pre-heap and a closed command (i.e. a command with no free

variables) to an outcome γ, which is either of the form (n, v, h') where $n \in \mathbb{N}$ is the number of execution steps performed, v is the result value, and h' is the post-heap, or an *exception* E, which is either $\mathbf{Failure}(n)$, where $n \in \mathbb{N}$ is the number of execution steps performed, or $\mathbf{Divergence}$. We define $n + \gamma$ as follows: $n + (n', v, h') = (n + n', v, h')$; $n + \mathbf{Failure}(n') = \mathbf{Failure}(n + n')$; $n + \mathbf{Divergence} = \mathbf{Divergence}$. We define the big-step relation coinductively [4] by means of the rules shown in Figure 4.

$$(h, v) \Downarrow (1, v, h) \qquad \frac{(h, c) \Downarrow (n, v, h') \qquad (h', c'[v/x]) \Downarrow \gamma}{(h, \tau\ x := c; c') \Downarrow n + \gamma} \qquad \frac{(h, c) \Downarrow E}{(h, \tau\ x := c; c') \Downarrow 1 + E}$$

$$\frac{\mathbf{class}\ C \cdots \{ \cdots\ \mathbf{static}\ \tau\ m(\overline{\tau\ x})\ \{\ c\ \}\ \cdots \} \qquad (h, c[\overline{v}/\overline{x}]) \Downarrow \gamma}{(h, C.m(\overline{v})) \Downarrow 1 + \gamma}$$

$$\frac{\begin{array}{c}\mathsf{classOf}(o) = C \\ \mathbf{class}\ C \cdots \{ \cdots\ \mathbf{instance}\ \tau\ m(\overline{\tau\ x})\ \{\ c\ \}\ \cdots \} \qquad (h, c[o, \overline{v}/\mathsf{this}, \overline{x}]) \Downarrow \gamma\end{array}}{(h, o.m(\overline{v})) \Downarrow 1 + \gamma}$$

$$\frac{\mathsf{classOf}(o) = C \qquad \mathbf{class}\ C(\overline{\tau\ f})\ \cdots \qquad h' = h \uplus \{\overline{o.f \mapsto v}\}}{(h, \mathbf{new}\ C(\overline{v})) \Downarrow (1, o, h')}$$

$$\frac{(o, f) \in \mathrm{dom}(h)}{(h, o.f) \Downarrow (1, h((o, f)), h)} \qquad \frac{(o, f) \notin \mathrm{dom}(h)}{(h, o.f) \Downarrow \mathbf{Failure}(1)}$$

$$\frac{(o, f) \in \mathrm{dom}(h)}{(h, o.f := v) \Downarrow (1, v, h[(o, f) := v])} \qquad \frac{(o, f) \notin \mathrm{dom}(h)}{(h, o.f := v) \Downarrow \mathbf{Failure}(1)}$$

Fig. 4. Coinductive big-step semantics of the programming language

Note that $h \uplus h'$ is undefined if $\mathrm{dom}(h) \cap \mathrm{dom}(h') \neq \emptyset$.

We now define the meaning of assertions. To interpret an assertion, we need an interpretation for the predicates it uses. A predicate interpretation I is a set $I \subseteq \mathit{ObjRefs} \times \mathit{PredNames} \times \mathit{Values}^* \times \mathit{Heaps}$. If $(o, p, \overline{v}, h) \in I$, this means that according to interpretation I, predicate assertion $o.p(\overline{v})$ is true in heap h. We now define the truth $I, h \vDash P$ of a closed assertion P under a predicate

interpretation I and a heap h:

$$\begin{array}{lcl}
I, h \vDash v & \Leftrightarrow & v = \mathsf{true} \\
I, h \vDash o.f \mapsto v & \Leftrightarrow & (o, f) \mapsto v \in h \\
I, h \vDash P * P' & \Leftrightarrow & \exists h_1, h_2.\ h = h_1 \uplus h_2 \wedge I, h_1 \vDash P \wedge I, h_2 \vDash P' \\
I, h \vDash P \wedge P' & \Leftrightarrow & I, h \vDash P \wedge I, h \vDash P' \\
I, h \vDash P \vee P' & \Leftrightarrow & I, h \vDash P \vee I, h \vDash P' \\
I, h \vDash o.p(\overline{v}) & \Leftrightarrow & (o, p, \overline{v}, h) \in I
\end{array}$$

We can now define a function $F(I) = \{(o, p, \overline{v}, h) \mid I \vDash (o, p, \overline{v}, h)\}$ that produces a predicate interpretation, using I for nested predicate assertions, where

$$\frac{\begin{array}{c}\mathsf{classOf}(o) = C \qquad \textbf{class}\ C \cdots\ \{\ \cdots\ \textbf{predicate}\ p(\overline{\tau\ x}) = P\ \cdots\ \} \\ \overline{y} = \mathsf{FV}(P[\overline{v}/\overline{x}]) \qquad I, h \vDash P[\overline{v}/\overline{x}, \overline{w}/\overline{y}]\end{array}}{I \vDash (o, p, \overline{v}, h)}$$

Notice that free variables in a predicate body are implicitly existentially quantified.

It is easy to check that F is monotonic: $I \subseteq I' \Rightarrow F(I) \subseteq F(I')$. (This would not be the case if our assertion language included negation or implication of assertions.) Therefore, by the Knaster-Tarski theorem, $I_{\text{fix}} = \bigcap\{I \mid F(I) \subseteq I\}$ is the least fixpoint of F. We adopt I_{fix} as the meaning of predicates.

We are now ready to define the meaning of Hoare triples (for partial correctness):

$$\vDash \{P\}\ c\ \{Q\} \quad \Leftrightarrow \quad \forall h, \gamma.\ I_{\text{fix}}, h \vDash P \wedge (h, c) \Downarrow \gamma \Rightarrow \gamma \vDash Q$$

where satisfaction $\gamma \vDash Q$ of a postcondition by an outcome is defined as:

$$\textbf{Divergence} \vDash Q \qquad\qquad \frac{I_{\text{fix}}, h \vDash Q[v/\mathsf{res}]}{(n, v, h) \vDash Q}$$

The proof rules are shown in Figure 5. Notice that in method contracts, free variables in the precondition are universally quantified across the contract; their scope extends to the postcondition as well. Variables that are free only in the postcondition are existentially quantified in the postcondition.

We consider a class to implement an interface method if the class has a method of the same name, return type, and parameter list, whose body satisfies the interface method's contract. An alternative approach is to check *compatibility* of the class method's contract with the interface method's contract [5].

The rule of consequence uses validity of implications. We define $\vDash P \Rightarrow P'$ as $\forall h.\ I_{\text{fix}}, h \vDash P \Rightarrow I_{\text{fix}}, h \vDash P'$. In particular, we have

$$\frac{\textbf{class}\ C \cdots\ \{\ \cdots\ \textbf{predicate}\ p(\overline{\tau\ x}) = P;\ \cdots\ \}}{\begin{array}{l}\vDash \mathsf{classOf}(o) = C \wedge o.p(\overline{v}) \Rightarrow P[o/\mathsf{this}, \overline{v}/\overline{x}] \\ \vDash \mathsf{classOf}(o) = C \wedge P[o/\mathsf{this}, \overline{v}/\overline{x}] \Rightarrow o.p(\overline{v})\end{array}}$$

The proof rules are sound: $\vdash \{P\}\ c\ \{Q\} \Rightarrow \vDash \{P\}\ c\ \{Q\}$. By the following lemma:

$$\vdash \{\mathsf{true}\}\ v\ \{\mathsf{res} = v\} \qquad \frac{\vdash \{P\}\ c\ \{Q\} \qquad \forall v.\ \vdash \{Q[v/\mathsf{res}]\}\ c'[v/x]\ \{R\}}{\vdash \{P\}\ \tau\ x := c; c'\ \{R\}}$$

$$\frac{\begin{array}{c}\mathbf{class}\ C \cdots\ \{\ \cdots\ \mathbf{static}\ \tau\ m(\overline{\tau\ x})\ \mathbf{req}\ P;\ \mathbf{ens}\ Q;\ \cdots\ \} \\ \overline{y} = \mathsf{FV}(P) \setminus \overline{x} \qquad \overline{z} = \mathsf{FV}(Q) \setminus \overline{x}, \mathsf{result}, \overline{y}\end{array}}{\vdash \{P[\overline{v}/\overline{x}, \overline{w}/\overline{y}]\}\ C.m(\overline{v})\ \{\exists \overline{w'}.\ Q[\overline{v}/\overline{x}, \overline{w}/\overline{y}, \overline{w'}/\overline{z}, \mathsf{res}/\mathsf{result}]\}}$$

$$\frac{\begin{array}{c}\mathbf{interface}\ \iota\ \{\ \cdots\ \tau\ m(\overline{\tau\ x});\ \mathbf{req}\ P;\ \mathbf{ens}\ Q;\ \cdots\ \} \\ \overline{y} = \mathsf{FV}(P) \setminus \mathsf{this}, \overline{x} \qquad \overline{z} = \mathsf{FV}(Q) \setminus \mathsf{this}, \overline{x}, \mathsf{result}, \overline{y}\end{array}}{\vdash \{P[o/\mathsf{this}, \overline{v}/\overline{x}, \overline{w}/\overline{y}]\}\ o.m(\overline{v})\ \{\exists \overline{w'}.\ Q[o/\mathsf{this}, \overline{v}/\overline{x}, \overline{w}/\overline{y}, \overline{w'}/\overline{z}, \mathsf{res}/\mathsf{result}]\}}$$

$$\frac{\mathbf{class}\ C(\overline{\tau\ f})\ \cdots}{\{\mathsf{true}\}\ \mathbf{new}\ C(\overline{v})\ \{\odot_{(f,v)\in\overline{(f,v)}}\ \mathsf{res}.f \mapsto v\}} \qquad \{o.f \mapsto v\}\ o.f\ \{o.f \mapsto v \wedge \mathsf{res} = v\}$$

$$\{o.f \mapsto \_\}\ o.f := v\ \{o.f \mapsto v\} \qquad \frac{\vDash P \Rightarrow P' \qquad \vdash \{P'\}\ c\ \{Q'\} \qquad \vDash Q' \Rightarrow Q}{\vdash \{P\}\ c\ \{Q\}}$$

$$\frac{\vdash \{P\}\ c\ \{Q\}}{\vdash \{P * R\}\ c\ \{Q * R\}} \qquad \frac{\vdash \{P\}\ c\ \{Q\} \qquad \vdash \{P'\}\ c\ \{Q\}}{\vdash \{P \vee P'\}\ c\ \{Q\}}$$

$$\frac{\mathit{program} = \overline{\mathit{tdef}} \qquad \overline{\vdash \mathit{tdef}\ \mathsf{ok}}}{\vdash \mathit{program}\ \mathsf{ok}} \qquad \vdash \mathit{idef}\ \mathsf{ok}$$

$$\frac{\overline{C \vdash \mathit{cmdef}\ \mathsf{ok}} \qquad \mathbf{interface}\ \iota\ \{\ \overline{\mathit{pdecl}}\ \overline{\mathit{imdef}}\ \} \qquad \overline{\vdash C\ \mathsf{implements}\ \mathit{imdef}}}{\vdash \mathbf{class}\ C(\cdots)\ \mathbf{implements}\ \iota\ \{\ \overline{\mathit{pdef}}\ \overline{\mathit{cmdef}}\ \}\ \mathsf{ok}}$$

$$\frac{\begin{array}{c}\overline{y} = \mathsf{FV}(P) \setminus \overline{x} \qquad \overline{z} = \mathsf{FV}(Q) \setminus \overline{x}, \mathsf{result}, \overline{y} \\ \forall \overline{v}, \overline{w}.\ \vdash \{P[\overline{v}/\overline{x}, \overline{w}/\overline{y}]\}\ c[\overline{v}/\overline{x}]\ \{\exists \overline{w'}.\ Q[\overline{v}/\overline{x}, \mathsf{res}/\mathsf{result}, \overline{w}/\overline{y}, \overline{w'}/\overline{z}]\}\end{array}}{C \vdash \mathbf{static}\ \tau\ m(\overline{\tau\ x})\ \mathbf{req}\ P;\ \mathbf{ens}\ Q;\ \{\ c\ \}\ \mathsf{ok}}$$

$$\frac{\begin{array}{c}\overline{y} = \mathsf{FV}(P) \setminus \mathsf{this}, \overline{x} \qquad \overline{z} = \mathsf{FV}(Q) \setminus \mathsf{this}, \overline{x}, \mathsf{result}, \overline{y} \\ \forall o, \overline{v}, \overline{w}.\ \vdash \{P[o/\mathsf{this}, \overline{v}/\overline{x}, \overline{w}/\overline{y}]\}\ c[o/\mathsf{this}, \overline{v}/\overline{x}]\ \{\exists \overline{w'}.\ Q[o/\mathsf{this}, \overline{v}/\overline{x}, \mathsf{res}/\mathsf{result}, \overline{w}/\overline{y}, \overline{w'}/\overline{z}]\}\end{array}}{C \vdash \mathbf{instance}\ \tau\ m(\overline{\tau\ x})\ \mathbf{req}\ P;\ \mathbf{ens}\ Q;\ \{\ c\ \}\ \mathsf{ok}}$$

$$\frac{\begin{array}{c}\mathbf{class}\ C(\cdots)\ \cdots\ \{\ \cdots\ \mathbf{instance}\ \tau\ m(\overline{\tau\ x})\ \mathbf{req}\ P';\ \mathbf{ens}\ Q';\ \{\ c\ \}\ \cdots\ \} \\ \overline{y} = \mathsf{FV}(P) \setminus \mathsf{this}, \overline{x} \qquad \overline{z} = \mathsf{FV}(Q) \setminus \mathsf{this}, \overline{x}, \mathsf{result}, \overline{y} \\ \forall o, \overline{v}, \overline{w}.\ \vdash \{P[o/\mathsf{this}, \overline{v}/\overline{x}, \overline{w}/\overline{y}]\}\ c[o/\mathsf{this}, \overline{v}/\overline{x}]\ \{\exists \overline{w'}.\ Q[o/\mathsf{this}, \overline{v}/\overline{x}, \mathsf{res}/\mathsf{result}, \overline{w}/\overline{y}, \overline{w'}/\overline{z}]\}\end{array}}{\vdash C\ \mathsf{implements}\ \tau\ m(\overline{\tau\ x});\ \mathbf{req}\ P;\ \mathbf{ens}\ Q;}$$

Fig. 5. Proof rules of the program logic for partial correctness

Lemma 1.

$$\begin{aligned}&\forall n, P, c, Q.\ \vdash \{P\}\ c\ \{Q\} \Rightarrow \\ &\quad \forall h, h_0, h_F, \gamma.\ h = h_0 \uplus h_F \wedge I_{\text{fix}}, h_0 \vDash P \wedge (h, c) \Downarrow \gamma \Rightarrow \\ &\qquad \gamma \neq \textbf{Failure}(n) \\ &\qquad \wedge\ (\forall h', v.\ \gamma = (n, v, h') \Rightarrow \exists h_0'.\ h' = h_0' \uplus h_F \wedge I_{\text{fix}}, h_0' \vDash Q[v/\mathsf{res}])\end{aligned}$$

Proof. By well-founded induction on n. Fix some n_0 and assume the lemma holds for all $n < n_0$. We prove that it holds for $n = n_0$. By nested induction on the derivation of $\vdash \{P\}\ c\ \{Q\}$. □

3 Termination of Sequential Programs

How to extend this program logic so that it verifies the absence of infinite recursion? We wish to impose an additional proof obligation at method call sites, such that during any program execution only a finite number of method calls occur. Since we are already using separation logic, which can be interpreted as a logic of permissions, we introduce the notion of *call permissions*. If we make available to a program's main method only a finite stock of call permissions, and a call permission is consumed at each call site, then it follows that an execution can perform only finitely many calls.

Note that we should not count call permissions merely using a natural number. This would mean each method's specification would state an upper bound on the number of calls it performs. That would seem to require much tedious and brittle bookkeeping, and cause problems if the number of calls depends on nondeterministic phenomena such as user input.

The well-known solution to the counting issue in termination proofs is the use of *well-founded relations*. A well-founded relation is one that admits no infinite descending chains, or, equivalently, where each nonempty set has a minimal element. In this paper, we will more specifically use *ordinals*, well-founded relations that are additionally strict total orders, and for which useful conventional notations exist.[3][4]

We briefly review the ordinal theory used in this paper. The *finite ordinals* are the natural numbers, with their usual order. The set of finite ordinals is denoted ω. The *product* $\alpha \cdot \beta$ of two sets of ordinals is the set of pairs $(a, b) \in \alpha \times \beta$, with their *lexicographical ordering* (with the least significant element first): $(a, b) < (a', b')$ iff $b < b'$ or $b = b$ and $a < a'$. The *exponentiation* α^β of two sets of ordinals is the set of functions $f : \beta \to \alpha$ where only finitely many arguments map to nonzero values; the order is a generalization of the lexicographic order: $f < f'$ iff $f \neq f'$ and $f(b) < f'(b)$ where b is the maximum

[3] Our implementation of the proposed proof system (see Sec. 6) supports arbitrary well-founded relations.

[4] Technically, the ordinals are the equivalence classes of well-ordered sets under isomorphism. A well-ordered set is a set with a well-ordering, i.e. a well-founded strict total order. In an abuse of terminology, we will identify each such equivalence class with each of its members.

argument such that $f(b) \neq f'(b)$. In particular, ω^X yields the *multisets (or bags)* of elements of X, with *multiset order*. We denote bags using fat braces: $\{\!|a, b, c|\!\} = \mathbf{0} \uplus \{\!|a|\!\} \uplus \{\!|b|\!\} \uplus \{\!|c|\!\}$, where $\mathbf{0}$ denotes the empty multiset: $\mathbf{0} = \lambda x.\ 0$, and $M \uplus M'$ denotes multiset union: $M \uplus M' = (\lambda x.\ M(x) + M'(x))$. In order to descend down the multiset order starting from a multiset M, one can replace any element of M with any number of lesser elements of X, any number of times. For example, $\{\!|0, 0, 1, 2, 2, 2|\!\} < \{\!|0, 0, 0, 3|\!\}$.

Our program logic is based on the notion that at each point during a program's execution, it has a stock of call permissions in the form of a *bag of ordinals* $\Lambda \in \omega^{Ordinals}$ (for some fixed set of ordinals *Ordinals*). We admit ghost execution steps that reduce the stock of call permissions to a lesser one. Furthermore, at each call, an element is removed from the bag. It follows that the program terminates: an infinite execution would constitute an infinite descending chain in $\omega^{Ordinals}$.

We extend our separation logic with an assertion for call permissions:

$$P ::= o.f \mapsto v \mid P * P \mid \mathsf{call\_perm}(\alpha) \mid \cdots$$

We interpret assertions under a predicate interpretation, a heap and a stock of call permissions:

$$\begin{array}{lcl} I, h, \Lambda \vDash \mathsf{call\_perm}(\alpha) & \Leftrightarrow & \alpha \in \Lambda \\ I, h, \Lambda \vDash P * P' & \Leftrightarrow & \exists h_1, \Lambda_1, h_2, \Lambda_2.\ h = h_1 \uplus h_2 \wedge \Lambda = \Lambda_1 \uplus \Lambda_2 \\ & & \wedge\ I, h_1, \Lambda_1 \vDash P \wedge I, h_2, \Lambda_2 \vDash P' \end{array}$$

The meaning of Hoare triples is now defined as follows:

$$\vDash \{P\}\ c\ \{Q\} \quad \Leftrightarrow \quad \forall h, \Lambda, \gamma.\ I_{\text{fix}}, h, \Lambda \vDash P \wedge (h, c) \Downarrow \gamma \Rightarrow \gamma \vDash_\Lambda Q$$

where satisfaction $\gamma \vDash_\Lambda Q$ of a postcondition by an outcome under a given stock of call permissions is now defined as follows:

$$\frac{\Lambda' \leq \Lambda \qquad I_{\text{fix}}, h, \Lambda' \vDash Q[v/\mathsf{res}]}{(n, v, h) \vDash_\Lambda Q}$$

Notice that divergence is no longer considered to satisfy a postcondition.

The only proof rules that change are the rule of consequence and the rules for method calls. For the rule of consequence of our logic, we use a notion of implication that allows weakening of the stock of call permissions:

$$P \sqsubseteq P' \Leftrightarrow \forall h, \Lambda.\ h, \Lambda \vDash P \Rightarrow \exists \Lambda' < \Lambda.\ h, \Lambda' \vDash P'$$

We then have $\mathsf{call\_perm}(1) \sqsubseteq \mathsf{call\_perm}(0) * \mathsf{call\_perm}(0)$, and, more generally, $\mathsf{call\_perm}(1) \sqsubseteq \odot_{i=1}^{n} \mathsf{call\_perm}(0)$, for any n, where $\odot_{i=a}^{b} P(i)$ represents iterated separating conjunction:

$$\odot_{i=a}^{b} P(i) = \begin{cases} \mathsf{true} & \text{if } b < a \\ P(a) * \odot_{i=a+1}^{b} & \text{otherwise} \end{cases}$$

In case i does not appear in P, we abbreviate $\odot_{i=1}^{n} P$ to $n \cdot P$. So, for any $\alpha' < \alpha$ and any n, we have $\mathsf{call\_perm}(\alpha) \sqsubseteq n \cdot \mathsf{call\_perm}(\alpha')$.

The proof rules for method calls are as follows:

$$\frac{\begin{array}{c}\textbf{class } C \cdots \{ \cdots \textbf{ static } \tau\ m(\overline{\tau\ x})\ \textbf{req } P;\ \textbf{ens } Q;\ \cdots \} \\ \overline{y} = \mathsf{FV}(P) \setminus \overline{x} \qquad \overline{z} = \mathsf{FV}(Q) \setminus \overline{x}, \mathsf{result}, \overline{y}\end{array}}{\vdash \{\mathsf{call\_perm}(\_) * P[\overline{v}/\overline{x}, \overline{w}/\overline{y}]\}\ C.m(\overline{v})\ \{\exists \overline{w'}.\ Q[\overline{v}/\overline{x}, \overline{w}/\overline{y}, \overline{w'}/\overline{z}, \mathsf{res}/\mathsf{result}]\}}$$

$$\frac{\begin{array}{c}\textbf{interface } \iota\ \{ \cdots\ \tau\ m(\overline{\tau\ x}); \textbf{ req } P; \textbf{ ens } Q; \cdots \} \\ \overline{y} = \mathsf{FV}(P) \setminus \mathsf{this}, \overline{x} \qquad \overline{z} = \mathsf{FV}(Q) \setminus \mathsf{this}, \overline{x}, \mathsf{result}, \overline{y} \qquad \theta = o/\mathsf{this}, \overline{v}/\overline{x}, \overline{w}/\overline{y}\end{array}}{\vdash \{\mathsf{call\_perm}(\_) * P[\theta]\}\ o.m(\overline{v})\ \{\exists \overline{w'}.\ Q[\theta, \overline{w'}/\overline{z}, \mathsf{res}/\mathsf{result}]\}}$$

Soundness follows from the following lemma:

Lemma 2.

$$\begin{array}{l}\forall \Lambda, c, P, Q.\ \vdash \{P\}\ c\ \{Q\} \Rightarrow \\ \quad \forall h, h_0, h_F, \gamma.\ h = h_0 \uplus h_F \wedge I_{\mathrm{fix}}, h_0, \Lambda \vDash P \wedge (h, c) \Downarrow \gamma \Rightarrow \\ \quad\quad \exists n, h', h_0', v, \Lambda'.\ \gamma = (n, v, h') \wedge h' = h_0' \uplus h_F \wedge I_{\mathrm{fix}}, h_0', \Lambda' \vDash Q[v/\mathsf{res}]\end{array}$$

Proof. By well-founded induction on $(|c|, \Lambda)$, where $|c|$ is the syntactic size of command c. Fix some (N_0, Λ_0) and assume the lemma holds for all $(|c|, \Lambda) < (N_0, \Lambda_0)$. We prove that it holds for $(|c|, \Lambda) = (N_0, \Lambda_0)$. By nested induction on the derivation of $\vdash \{P\}\ c\ \{Q\}$. □

Now that we have call permissions, how do we use them to write modular specifications? Clearly, each method should require some call permissions from its caller in order to be able to perform calls itself. But how much? Which ordinal?

It should not be necessary for a method to ask specifically for call permissions for each call it performs. In particular, we assume a model where a program consists of layers where each layer is built on top of lower layers to offer functionality to higher layers. In this model, a method's contract should not reveal whether, or how often, the method calls into lower layers. This is clearly an implementation detail that should not concern the method's clients. To support this notion, we assume that class definitions that appear earlier in a program text constitute lower layers with respect to class definitions that appear later. Similarly, within a class definition, we assume that methods that appear earlier constitute lower layers with respect to methods that appear later. To enable abstraction over calls to lower-layer methods, we will use class methods as ordinals, ordered by their position in the program text.

Using this approach, if all calls in a program call static methods in lower layers, it is sufficient for each method $C.m$ to require $\mathsf{call\_perm}(C.m)$. Indeed, a valid proof outline is shown in Figure 6.

Methods main and sqrt, before performing their nested call, replace the incoming call permission qualified by their own name with two copies of a call permission qualified by their callee's name. One copy is consumed at the start of the call, the other is passed into the callee as required by its precondition.

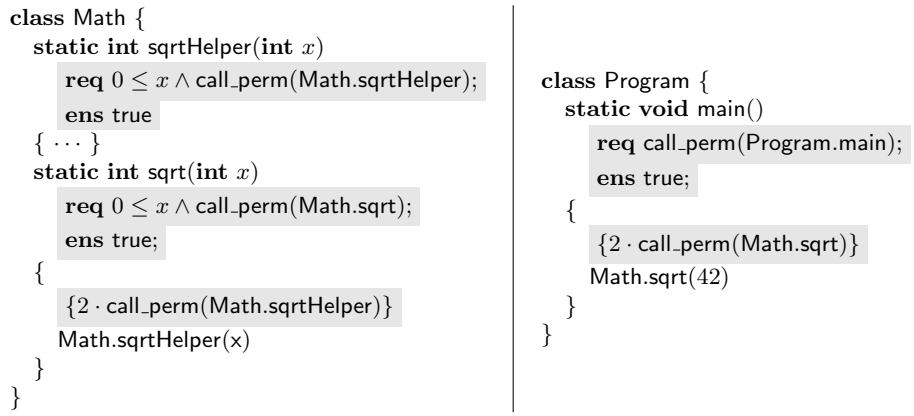

Fig. 6. A program with calls to lower-layer static methods only

Of course, in most programs not all calls, even if they call static methods, call lower-layer methods, especially since our programming language does not have loops. How to deal with recursive methods? First of all, we assume that lower layers are developed before higher layers, and therefore lower layers never call static methods in higher layers. It follows that each cycle in a program's graph of calls to static methods is contained entirely within a single module. In our approach, then, all such members of a recursive cycle should be private to the module, and therefore their contracts need not be abstract. Separate public methods should be provided that call into the cycle but are not part of it.

An example is shown in Figure 7.

Notice that each of the members of the cycle requires call permission for the maximum member. The contracts of isOddIter and isEvenIter are not abstract, but those of isOdd and isEven are.

Notice that the coefficient of the call permission in the contracts of the recursive methods serves the role of the classical recursion measure. However, sometimes a recursion measure cannot easily be expressed as a simple natural number.

To support such recursion measures, we move, for the set of ordinals that we use to qualify call permissions, from methods to pairs of *local ordinals* and methods (where the method is the most significant component), given some fixed set *LocOrd* of local ordinals. Public methods $C.m$ should request call_perm$((0, C.m))$.

```
class Math {
  static bool isOddIter(int x)
    req 0 ≤ x ∧
      x · call_perm(Math.isEvenIter);
    ens result = (x mod 2 = 1);
  {
    if x = 0 then false else
      Math.isEvenIter(x − 1)
  }
  static bool isEvenIter(int x)
    req 0 ≤ x ∧
      x · call_perm(Math.isEvenIter);
    ens result = (x mod 2 = 0);
  {
    if x = 0 then true else
      Math.isOddIter(x − 1)
  }
```

```
  static bool isOdd(int x)
    req 0 ≤ x ∧ call_perm(Math.isOdd);
    ens result = (x mod 2 = 0);
  { Math.isOddIter(x) }
  static bool isEven(int x)
    req 0 ≤ x ∧ call_perm(Math.isEven);
    ens result = (x mod 2 = 0);
  { Math.isEvenIter(x) }
}
```

Fig. 7. Recursion measured by a natural number

A classic example where an infinite ordinal is needed, is the Ackermann function:

```
class Math {
  static int ackermannIter(int m, int n)
    req 0 ≤ m ∧ 0 ≤ n ∧ call_perm(((m, n), Math.ackermannIter));
    ens result = Ack(m, n);
  {
    if n = 0 then m + 1
    else if m = 0 then Math.ackermannIter(1, n − 1)
    else {
      int r := Math.ackermannIter(m − 1, n);
      Math.ackermannIter(r, n − 1)
    }
  }
  static int ackermann(int m, int n)
    req 0 ≤ m ∧ 0 ≤ n ∧ call_perm((0, Math.ackermann));
    ens result = Ack(m, n);
  { Math.ackermannIter(m, n) }
}
```

We are now ready to consider the case of programs that call interface methods. Consider a method integrate for computing the integral of a real function

over an interval:

```
interface RealFunc {
  double apply(double x);
}
class Math {
  static double integrate(double a, double b, RealFunc f)
  { ··· }
}
```

What call permissions should method integrate request of its caller? Clearly, the method should be allowed to call method apply of object f. And it should be allowed to call it not just once, but arbitrarily often. Since the calls of f.apply might occur inside recursive helper functions measured by arbitrary ordinals, there is no single ordinal that can obviously serve as an upper bound. Furthermore, method integrate should be allowed to pass f to helper methods in lower layers, and those should themselves be specified abstractly without revealing how often they call f.

To solve this problem, we move, for the set of ordinals that we use to qualify call permissions, from pairs of local ordinals and methods to pairs of local ordinals and *bags of methods*:

$$\begin{aligned} \mathit{ClassMethods} &= \{C.m \mid \textbf{class } C \ \cdots \ \{ \ \cdots \tau \ m(\cdots) \ \cdots \ \} \\ \mathit{MethodBags} &= \omega^{\mathit{ClassMethods}} \\ \mathit{Ordinals} &= \mathit{LocOrd} \cdot \mathit{MethodBags} \end{aligned}$$

Note that set *ClassMethods* includes both the static methods and the instance methods.

Public methods $C.m$ in programs that do not call interface methods should request call_perm$((0, \{\!|C.m|\!\}))$.

The following contract for method integrate allows it to call f.apply arbitrarily often, and to delegate a similar permission to lower-layer static methods:

```
static double integrate(double a, double b, RealFunc f)
  req call_perm((0, {|Math.integrate, f.apply|}));
```

Note that we use $o.m$ as a shorthand for classOf$(o).m$.

However, we are not done. Indeed, when calling f.apply, we need not just the call permission that is consumed by the call itself, but also the call permissions requested by f.apply's precondition. What should those be?

f.apply should be allowed to call static methods at layers below itself, so it should at least receive a call permission qualified by its own name. However, there are other methods that it should be allowed to call as well. Indeed, through the fields of its this object, i.e. through the fields of f, this method may be able to reach directly or indirectly any number of objects and might need to perform any number of calls on any number of methods thereof. The method should request permission for those calls as well.

But how can it abstractly request permission to call these methods, hidden inside its private data structures, with whose existence its clients should not otherwise be concerned?

We solve this problem by allowing the bag of methods reachable from an object to be named abstractly by exposing it as an argument of the predicate family that describes the object.

Consider first the partial-correctness specification for interface RealFunc in Figure 8(a). We extend it for total corectness as shown in Figure 8(b).

```
interface RealFunc {
  predicate RealFunc();
  double apply(double x);
    req this.RealFunc();
    ens this.RealFunc();
}
```

(a) Partial correctness

```
interface RealFunc {
  predicate RealFunc(MethodBag d);
  double apply(double x);
    req this.RealFunc(d) ∗ call_perm((0, d));
    ens this.RealFunc(d);
}
```

(b) Total correctness

Fig. 8. Annotations for interface RealFunc

We adapt the contract of method integrate accordingly:

```
static double integrate(double a, double b, RealFunc f)
  req f.RealFunc(d) ∗ call_perm((0, {|Math.integrate|} ⊎ d));
```

A simple implementation of interface RealFunc is shown in Figure 9. Notice that

```
class LinearFunc(double a, double b) implements RealFunc {
  predicate RealFunc(MethodBag d) =
    this.a ↦ _ ∗ this.b ↦ _ ∧ d = {|this.apply|};
  double apply(double x)
  { double a := this.a; double b := this.b; a · x + b }
  static RealFunc createLinearFunc(double a, double b)
    req call_perm((0, {|LinearFunc.createLinearFunc|}));
    ens result.RealFunc(d) ∧ d < {|LinearFunc.createLinearFunc|};
  { new LinearFunc(a, b) }
}
```

Fig. 9. A simple implementation of interface RealFunc

method createLinearFunc's postcondition provides an upper bound on d. This enables the caller (who is necessarily in a higher layer than createLinearFunc) to

produce the call permissions required to call result.apply. Notice also that this upper bound does not constrain method createLinearFunc's implementation, if we assume that a method only allocates (through **new**) objects of classes defined in its own layer or in lower layers.

A slightly more involved implementation is shown in Figure 10. Notice that

```
class Sum(RealFunc f1, RealFunc f2) implements RealFunc {
  predicate RealFunc(MethodBag d) =
    this.f1 ↦ f1 ∗ f1.RealFunc(d1) ∗ this.f2 ↦ f2 ∗ f2.RealFunc(d2)
    ∧ d = {this.apply} ⊎ d1 ⊎ d2;
  double apply(double x) {
    RealFunc f1 := this.f1; RealFunc f2 := this.f2;
    double r1 := f1.apply(x); double r2 := f2.apply(x);
    r1 + r2
  }
  static RealFunc createSum(RealFunc f1, RealFunc f2)
    req f1.RealFunc(d1) ∗ f2.RealFunc(d2)
      ∗ call_perm((0, {Sum.createSum} ⊎ d1 ⊎ d2));
    ens result.RealFunc(d) ∧ d < {Sum.createSum} ⊎ d1 ⊎ d2;
  { new Sum(f1, f2) }
}
```

Fig. 10. An implementation of interface RealFunc

class Sum's instance of predicate family RealFunc defines its d parameter (which we call the *dynamic depth* since it gives a measure of the number of layers of abstraction of which the object is composed) as the multiset union of its own apply method and the referenced objects' dynamic depths. Indeed, as a general pattern, an object's dynamic depth should typically be defined as the union of its own methods and the dynamic depths of the objects stored in its fields. This allows the object to call those objects' methods, assuming their contracts follow the standard pattern exemplified by the contract of RealFunc.apply.

Notice also how this definition enables the successful verification of method apply.

Method createSum's precondition follows the general pattern: request a call permission qualified by a multiset that is the union of the method itself and the dynamic depths of any objects being passed into the method. Its postcondition follows a general pattern for methods that return a new object: the new object's dynamic depth is bounded by the same multiset used to qualify the call permission in the precondition. It follows that any caller of this method can also call the new object's methods.

An implementation of method integrate and an example client program are shown in Figure 11. The proof outline for method main starts by reducing the incoming call permission to twelve copies of a call permission that is greater than

```
class Math {
  static double integrateIter(double a, double dx, int n, RealFunc f)
    req 0 ≤ n ∧ f.RealFunc(d) ∗ call_perm((n, {Math.integrateIter} ⊎ d));
    ens f.RealFunc(d);
  {
    if n = 0 then 0 else {
      double y := f.apply(a);
      double ys := Math.integrateIter(a + dx, dx, n − 1, f);
      y × dx + ys
    }
  }
  static double integrate(double a, double b, RealFunc f)
  { Math.integrateIter(a, (b − a)/1000, 1000, f) }
}
class Program {
  static main()
    req call_perm((0, {Program.main}));
    ens true;
  {
    {12 · call_perm((0, {2 · LinearFunc, 2 · Sum, Math}))}
    RealFunc f1 := LinearFunc.createLinearFunc(2, 3);
    RealFunc f2 := LinearFunc.createLinearFunc(5, 6);
    RealFunc f3 := LinearFunc.createLinearFunc(5, 6);
    RealFunc f4 := Sum.createSum(f1, f2);
    RealFunc f5 := Sum.createSum(f3, f4);
    Math.integrate(0, 100, f5)
  }
}
```

Fig. 11. An implementation of method integrate and a client program

each of the call permissions required for the six calls and the six preconditions. (We use a class name as an abbreviation for its greatest method.) Indeed, we have the inequalities shown in Figure 12.

An example of an interface method that takes as an argument another object is shown in Figure 13.

Our approach supports programs written in continuation-passing style (CPS). To illustrate this, we add CPS versions of methods contains and intersects to the IntSet example. See Figures 14, 15, and 16. The predicate definitions and constructor methods remain unchanged and are not repeated.

Notice that the continuation interfaces do not follow the general specification pattern. Indeed, since the invoke methods are invoked only once, any call permissions they need can be passed via the ContainsCont or IntersectsCont predicate, respectively.

Notice that InvokeCont1.invoke and InvokeCont2.invoke each require a single call permission in order to perform the nested invoke call. It is passed via the

$$
\begin{aligned}
\mathsf{d1}, \mathsf{d2}, \mathsf{d3} &< \{\mathsf{LinearFunc.createLinearFunc}\} \\
\mathsf{d4} &< \{\mathsf{Sum.createSum}\} \uplus \mathsf{d1} \uplus \mathsf{d2} \\
&< \{\mathsf{Sum.createSum}, \mathsf{LinearFunc.createLinearFunc}\} \\
\mathsf{d5} &< \{\mathsf{Sum.createSum}\} \uplus \mathsf{d3} \uplus \mathsf{d4} \\
&< \{2 \cdot \mathsf{LinearFunc.createLinearFunc}, 2 \cdot \mathsf{Sum.createSum}\}
\end{aligned}
$$

Fig. 12. Inequalities relevant for the proof of the integrate client program

predicate. The ordinal is irrelevant and is existentially quantified. In contrast, InvokeCont3.invoke needs to call set.intersectsCPS and for that needs a properly qualified call permission. The call permission that the InvokeCont3 object needs to pass to the InvokeCont2 object can be derived from it.

4 Termination of Multithreaded Programs

In this section, we extend our approach to multithreaded programs. That is, we extend our programming language with forking and joining of threads, and with locks and channels (with unbounded buffering) for inter-thread communication. We do not consider condition variables.

In this setting, termination consists of two aspects: absence of infinite executions, and absence of deadlocks.

Our approach, as presented in the preceding section, already prevents infinite executions, provided that, if thread creation involves method calls, a call permission is consumed for each such method call, just like for other method calls. When a thread creates a new thread, the creating thread's stock of call permissions is split among the two threads. Furthermore, threads can exchange call permissions through synchronization constructs, just like they can exchange regular memory access permissions. Absence of infinite executions follows from the fact that the total stock of call permissions in the system, i.e. the multiset union of all threads' stocks of call permission (plus any call permissions owned by synchronization constructs while being transferred between threads) decreases at each method call.

We define a deadlock as an execution where a thread is eventually disabled forever, by being blocked in a synchronization construct and never getting unblocked. To verify absence of deadlocks, we adopt the approach of Leino et al. [6] for modular specification and verification of deadlock-freedom of programs involving channels, locks, and joining of threads. They present their approach in the context of their Chalice verification system, but since Chalice is, like separation logic, centered around the concept of *permissions*, porting the approach to separation logic is fairly straightforward.

In this approach, performing a **receive** operation on a channel consumes a *credit* for this channel, which can be thought of as a *receive permission*. At any point in time, the total number of credits for a channel in the system equals the number of messages in the channel's buffer plus the number of *debits* for that channel. A thread holding a debit for a channel has an obligation to send on

```
interface IntSet {
  predicate IntSet(MethodBag d);
  bool contains(int x);
    req this.IntSet(d) ∗ call_perm((0, d));
    ens this.IntSet(d);
  bool intersects(IntSet other);
    req this.IntSet(d) ∗ other.IntSet(do) ∗ call_perm((0, d ⊎ do));
    ens this.IntSet(d) ∗ other.IntSet(do);
}
class Empty() implements IntSet {
  predicate IntSet(MethodBag d) = d = {this.∗};
  bool contains(int x) { false }
  bool intersects(IntSet other) { false }
  IntSet createEmpty()
    req call_perm((0, {Empty.createEmpty}));
    ens result.IntSet(d) ∧ d < {Empty.createEmpty};
  { new Empty() }
}
class Insert(int elem, IntSet set) implements IntSet {
  predicate IntSet(MethodBag d) =
    this.elem ↦ elem ∗ this.set ↦ set ∗ set.IntSet(ds) ∧ d = {this.∗} ⊎ ds;
  bool contains(int x) {
    int elem := this.elem;
    if x = elem then true else {
      IntSet set := this.set;
      set.contains(x)
    }
  }
  bool intersects(IntSet other) {
    int elem := this.elem;
    bool contains := other.contains(elem);
    if contains then true else {
      IntSet set := this.set;
      set.intersects(other)
    }
  }
  IntSet createInsert(int elem, IntSet set)
    req set.IntSet(ds) ∗ call_perm((0, {Insert.createInsert} ⊎ ds));
    ens result.IntSet(d) ∧ d < {Insert.createInsert} ⊎ ds;
  { new Insert(elem, set) }
}
```

Fig. 13. An interface method that takes as an argument another object

```
interface ContainsCont {
  predicate ContainsCont(IntSet set, MethodBag d);
  void invoke(bool result);
    req this.ContainsCont(set, d) ∗ set.IntSet(d);
}
interface IntersectsCont {
  predicate IntersectsCont(IntSet set, MethodBag d, IntSet other, MethodBag do);
  void invoke(bool result);
    req this.IntersectsCont(set, d, other, do) ∗ set.IntSet(d) ∗ other.IntSet(do);
}
interface IntSet {
  bool containsCPS(int x, ContainsCont cont);
    req this.IntSet(d) ∗ call_perm((0, d)) ∗ cont.ContainsCont(this, d);
  bool intersectsCPS(IntSet other, IntersectsCont cont);
    req this.IntSet(d) ∗ other.IntSet(do) ∗ call_perm((0, d ⊎ do))
      ∗ cont.ContainsCont(this, d, other, do);
}
```

Fig. 14. Continuation-passing-style versions of contains and intersects. All postconditions are false and are not shown.

that channel. When a **receive** operation blocks, the receiving thread is effectively waiting for the thread holding the debit to perform the promised **send** operation. A thread may at any time create a debit. This ghost operation also creates a credit. A credit is also created by sending. A credit can be used to receive or to cancel out a debit.

A deadlock can occur only if eventually there is a cycle of threads, each of which is waiting on the next to terminate, to send on a channel, or to release a lock. This is prevented by assigning to each thread, channel, and lock (i.e. each potential target of a wait operation) a *wait level*, and by allowing a thread to perform a wait operation only if the wait level of the target is less than the wait levels of the thread's obligations (i.e. its own wait level as a target for join operations, the wait levels of the channels for which it holds debits, and the wait levels of the locks which it holds), assuming that the less-than relation on wait levels is a strict partial order.[5]

An issue not addressed by Leino et al. [6] is the matter of information hiding in method contracts: how to write contracts that do not unnecessarily reveal information about, or constrain, implementations? In particular, their approach requires that each method that directly or indirectly performs a wait operation on a pre-existing synchronization object specify in its precondition a constraint on the thread's *obligation set*, the set of the wait levels of its obligations. In

[5] Leino et al. [6] require the wait level to be *greater*.

```
class Empty() implements IntSet {
  bool containsCPS(int x, ContainsCont cont)
  { cont.invoke(false) }
  bool intersectsCPS(IntSet other, IntersectsCont cont);
  { cont.invoke(false) }
}
class InsertCont1(ContainsCont cont) implements ContainsCont {
  predicate ContainsCont(IntSet set, MethodBag d) =
    this.cont ↦ cont ∗ cont.ContainsCont(set0, d0)
    ∗ set0.elem ↦ _ ∗ set0.set ↦ set
    ∗ call_perm(_) ∧ d = {set0.∗} ⊎ d0;
  void invoke(bool result)
  { ContainsCont cont := this.cont; cont.invoke(result) }
}
class InsertCont2(IntersectsCont cont) implements IntersectsCont {
  predicate IntersectsCont(
      IntSet set, MethodBag d, IntSet other, MethodBag do) =
    this.cont ↦ cont ∗ cont.IntersectsCont(set0, d0, other, do)
    ∗ set0.elem ↦ _ ∗ set0.set ↦ set
    ∗ call_perm(_) ∧ d = {set0.∗} ⊎ d0;
  void invoke(bool result)
  { IntersectsCont cont := this.cont; cont.invoke(result) }
}
class InsertCont3(Insert set0, IntSet other, IntersectsCont cont)
  implements ContainsCont {
  predicate ContainsCont(IntSet set, MethodBag d) =
    this.set0 ↦ set0 ∗ this.other ↦ set ∗ this.cont ↦ cont
    ∗ (set0.IntSet(ds0) ∧ ds0 = {set0.∗} ⊎ ds1)
    ∗ call_perm((0, {InsertCont3.invoke} ⊎ ds1 ⊎ d))
    ∗ cont.IntersectsCont(set0, ds0, set, d);
  void invoke(bool result) {
    Insert set0 := this.set0; IntSet other := this.other;
    IntersectsCont cont := this.cont;
    if result then cont.invoke(true) else {
      IntSet set := set0.set;
      IntersectsCont cont1 := new InsertCont2(cont);
      set.intersectsCPS(other, cont1)
    }
  }
}
```

Fig. 15. Implementations of containsCPS and intersectsCPS in classes Empty and Insert (Part 1 of 2)

```
class Insert(int elem, IntSet set) implements IntSet {
  bool containsCPS(int x, ContainsCont cont) {
    int elem := this.elem;
    if x = elem then cont.invoke(true) else {
      IntSet set := this.set;
      ContainsCont cont1 := new InsertCont1(cont);
      set.containsCPS(x, cont1)
    }
  }
  bool intersectsCPS(IntSet other, IntersectsCont cont);
    int elem := this.elem;
    ContainsCont cont1 := new InsertCont3(this, other, cont);
    other.containsCPS(elem, cont1);
  }
}
```

Fig. 16. Implementations of containsCPS and intersectsCPS in classes Empty and Insert (Part 2 of 2)

particular, any such method must require of its caller that the obligation set's elements be above the wait level of the object it will wait on.

If the wait operation is part of the method's advertised behavior, and the caller is aware of the synchronization object's existence, then this is not a problem. But how to deal with *private* synchronization objects protecting a module's internal state, with which callers should not be concerned?

To address this problem, we propose to use the same *bags of methods* we use to qualify call permissions for verifying absence of infinite recursion, also to qualify wait levels. Specifically, we propose to use as the set of wait levels the set $\mathit{LocalLevels} \cdot \mathit{MethodBags}$, for some set *LocalLevels* of local levels used to order the various synchronization objects used within a particular module. We assume *LocalLevels* has a top element $\top$. A lock that protects a module C's static state (i.e. state not associated with a particular object) should have as its wait level the pair $(\ell, \{\!|C|\!\})$ for some ℓ (where, again we use a class name to abbreviate the class' greatest method). A lock that protects an object with dynamic depth d should have wait level (ℓ, d).

Each module's public methods, regardless of whether their current implementations actually perform wait operations, should require that the current thread's obligations be greater than the receiver's dynamic depth. This specification style allows every module to acquire private locks, and furthermore it enables each module to satisfy its callees' analogous requirements.

4.1 Compare-and-swap loops

We now further extend our programming language with atomic machine instructions such as atomic loads, stores, and compare-and-swap operations. The latter, of the form $x := a.\mathsf{compareAndSwap}(o, n)$, where a is an object encapsulating an

atomic variable, atomically checks if a's value equals old value o, and, if so, sets it to new value n. The operation returns the value read. We assume the operations have sequentially consistent semantics.

Atomic machine instructions are used to write *lock-free* concurrent data structures, such as queues or stacks used to distribute work in parallel computing. A data structure is lock-free if, at any point, there exists a number N such that if any of the threads currently accessing the data structure is scheduled for N steps, then at least one of the threads accessing the data structure will complete the operation. Note that this is not the case for data structures that use locks, since if the thread holding the lock is not scheduled, no other thread can make progress.

Consider the example of a lock-free stack in Figure 17.

```
class Node(int value, Node next) {}
class Stack(Node head) {
  void pushIter(int value) {
    Node head := ⟨this.head⟩; Node n := new Node(value, head);
    Node head1 :=
      ⟨ Node head1 := this.head; if head1 = head then this.head := n; head1 ⟩;
    if head1 ≠ head then this.pushIter(value)
  }
  void push(int value) { this.pushIter(value) }
}
```

Fig. 17. A lock-free stack. (The pop operation is similar and is not shown.)

We only show the push operation; the pop operation is similar. We use the *atomic block* notation $\langle c \rangle$ to denote the command c executed atomically. The code exhibits a typical compare-and-swap loop: it reads a shared variable, computes a new value, and then attempts to install the new value, under the condition that the variable was not changed by another thread in the meantime. Otherwise, it tries again.

Our approach for verifying absence of infinite executions can be used to verify termination of programs involving compare-and-swap loops like the one above. A proof outline for the example program is shown in Figure 18.

The proof is based on the observation that whenever an operation has to try again, then some other concurrent operation has succeeded. The idea is then that the operation that succeeds supplies a call permission to each of the concurrent operations that it causes to fail, to enable them to try again. Since it is not known beforehand how many concurrent operations will be in progress at that time, method push passes a call permission qualified with ω (the first infinite ordinal) to pushIter. Note: a pushIter execution does not use this call permission for its own recursive calls; rather, it uses it only when it succeeds, to supply the

```
class GhostBag[T] {
  predicate GhostBag(Bag[T] elems);
  void add(T value);
    req this.GhostBag(elems);
    ens this.GhostBag(elems ⊎ {value}) ∗ this.GhostBagHandle(value);
  void remove(T value);
    req this.GhostBag(elems) ∗ this.GhostBagHandle(value);
    ens this.GhostBag(elems − {value}) ∧ value ∈ elems;
}
class Stack(Node head , GhostBag[Node] readers ) {
  static predicate nodes(Node n) =
    n = null ∨ n.value ↦ _ ∗ n.next ↦ next ∗ nodes(next);
  predicate SpaceInv() =
    this.head ↦ h ∗ nodes(h) ∗ this.readers ↦̄ readers ∗ readers.GhostBag(rs)
    ∗ call_perm((|{r ∈ rs | r ≠ h}|, {Stack.pushIter}));
  predicate Stack(d) = atomic_space(this.SpaceInv) ∗ this.readers ↦̄ _
    ∧ d = {Stack.push};
  void pushIter(int value)
    req [π]this.Stack(d) ∗ call_perm((ω, {Stack.pushIter}));
    ens [π]this.Stack(d);
  {
    GhostBag[Node] readers := this.readers;
    Node head := ⟨
      Node head := this.head; readers.add(head); head
    ⟩;
    { [π]atomic_space(this.SpaceInv) ∗
      this.readers ↦̄ readers ∗ readers.GhostBagHandle(head) }
    Node n := new Node(value, head);
    Node head1 := ⟨
      readers.remove(head);
      Node head1 := this.head;
      if head1 = head then this.head := n;
      head1
    ⟩;
    if head1 ≠ head then this.pushIter(value)
  }
  void push(int value)
    req [π]this.Stack(d) ∗ call_perm((0, d));
    ens [π]this.Stack(d);
  { this.pushIter(value) }
}
```

Fig. 18. Proof outline for the lock-free stack example

required call permissions to the concurrent operations. At that point, it reduces ω to the number of concurrent operations in progress at that time.

The proof uses *fractional permissions* [7, 8] to enable sharing. Fractional points-to permissions, that enable read-only sharing of memory locations, are denoted $\ell \overset{\pi}{\mapsto} v$ where π denotes a positive real number, or $\ell \overset{-}{\mapsto} v$ if the precise fraction is irrelevant. A fraction can be applied to a predicate using the syntax $[\pi]p()$; the meaning of this is simply the predicate body, with the fraction applied to each separating conjunct of it. The notation $p()$ abbreviates $[1]p()$.

To verify the atomic blocks, the proof uses the notion of an *atomic space*: similar to a lock, an atomic space has an *atomic space invariant* that describes resources that are being shared between multiple threads and that should only be accessed through atomic blocks. The ghost operation of creating an atomic space consumes the resources described by the atomic space invariant, and produces an *atomic space handle* $\mathsf{atomic\_space}(p)$, where p is the name of a predicate that serves as the atomic space invariant. The proof rules are as follows:

$$p() \sqsubseteq \mathsf{atomic\_space}(p)$$

$$[\pi_1 + \pi_2]\mathsf{atomic\_space}(p) \Leftrightarrow [\pi_1]\mathsf{atomic\_space}(p) * [\pi_2]\mathsf{atomic\_space}(p)$$

$$\frac{\{p() * P\}\ c\ \{p() * Q\}}{\{[\pi]\mathsf{atomic\_space}(p) * P\}\ \langle c \rangle\ \{[\pi]\mathsf{atomic\_space}(p) * Q\}}$$

The proof tracks the set of operations in progress (i.e. the threads that have performed the atomic read but have not yet performed the corresponding compare-and-swap) through a *ghost object* readers, an instance of *ghost class* GhostBag, whose specification is shown in Figure 18. The ghost bag is owned by the same atomic space that owns the stack's head field and the linked list of nodes (described by the nodes predicate). The GhostBagHandle predicate that a thread receives when it inserts an element into the ghost bag enables it to "remember" that it has an element in the ghost bag in between the atomic operations. The ghost bag in particular contains, for each operation in progress, the value of the head field that it read. The atomic space additionally holds, for each operation in progress whose head value is out of date, a call permission. As a failed operation removes its element from the ghost bag, it can also extract a call permission from the atomic space, enabling it to try again.

5 Liveness

Many programs, such as servers, are not supposed to terminate. Still, they should be *responsive*: if there are pending requests, the server should eventually respond. More generally, a program should always eventually interact with its environment; we call this *liveness*.

For example, consider the following program:

```
class OS {
  static void beep();
}
```

```
class Program {
  static void iter()
  { OS.beep(); Program.iter() }
  static void main()
  { Program.iter() }
}
```

This program does not terminate; however, it is live.

There is a simple way to encode a liveness property as a termination property. Suppose we wish to verify that a program always eventually performs an I/O operation. Then it is sufficient to prove that the program terminates, assuming that the N'th I/O operation causes the program to terminate, for some unknown but fixed N. This can be encoded into a specification of our approach as shown in Figure 19.

```
class OS {
  static predicate IO(int n);
  static void beep();
    req IO(n);
    ens 0 < n ∧ IO(n − 1);
}
```

```
class Program {
  static void iter()
    req IO(n) ∗ call_perm((n, {Program.iter}));
    ens false;
  { OS.beep(); Program.iter() }
  static void main()
    req IO(n) ∗ call_perm((0, {Program.main}));
    ens false;
  { Program.iter() }
}
```

Fig. 19. Liveness verification example

6 Implementation

We integrated the logic into the program verification tool VeriFast. Although VeriFast supports C and Java, for now we have added support for verification of termination only for C programs. We introduced the function specification clause **terminates**, to indicate that a function should terminate. In order to reduce specification overhead for functions that do not perform callbacks, our implementation offers a ghost command that allows a function to produce out of thin air any call permission whose bag of functions is less than itself (considered as a singleton bag). In exchange, our implementation consumes at a call site not just any call permission, but only a call permission whose function bag includes

the function being called:

$$\frac{f \in d \qquad \textbf{function}\ f(\overline{x})\ \textbf{req}\ P;\ \textbf{ens}\ Q;\ \{\ c\ \} \qquad \overline{y} = \mathsf{FV}(P) \setminus \overline{x} \qquad \overline{z} = \mathsf{FV}(Q) \setminus \overline{x}, \mathsf{result}, \overline{y}}{f_0 \vdash \{\mathsf{call\_perm}((\alpha, d)) * P[\overline{v}/\overline{x}, \overline{w}/\overline{y}]\}\ f(\overline{v})\ \{\exists \overline{w'}.\ Q[\overline{v}/\overline{x}, \overline{w}/\overline{y}, \overline{w'}/\overline{z}, \mathsf{res}/\mathsf{result}]\}}$$

$$\frac{\forall \overline{v}, \overline{w}.\ f \vdash \{P[\overline{v}/\overline{x}, \overline{w}/\overline{y}]\}\ c\ \{\exists \overline{w'}.\ Q[\overline{v}/\overline{x}, \overline{w}/\overline{y}, \overline{w'}/\overline{z}, \mathsf{res}/\mathsf{result}]\} \qquad \overline{y} = \mathsf{FV}(P) \setminus \overline{x} \qquad \overline{z} = \mathsf{FV}(Q) \setminus \overline{x}, \mathsf{result}, \overline{y}}{\vdash \textbf{function}\ f(\overline{x})\ \textbf{req}\ P;\ \textbf{ens}\ Q;\ \{\ c\ \}\ \mathsf{ok}}$$

$$\frac{d < \{\!\{f\}\!\}}{f \vdash \{\mathsf{true}\}\ \textbf{produce\_call\_perm}\ \{\mathsf{call\_perm}((\alpha, d))\}}$$

Our implementation is included in the latest VeriFast release, which is available at http://www.cs.kuleuven.be/~bartj/verifast/.

The distribution includes, in the directory examples/termination, the examples ackermann.c, simple_recursion.c (the isEven example), funcptr.c (corresponding to the IntFunc example of this paper), and cons.c (corresponding to the IntSet example). Furthermore, it includes, in the directory examples/termination/concurrent, specifications for semaphores (semas.h), channels (channels.h), and mutexes (mutexes.h), based on the ideas of Leino et al. [6], and the examples prodcons.c (a producer-consumer example using channels) and stack.c (the compare-and-swap loop example).

Porting our implementation to Java programs is mostly straightforward. One issue, however, is that in Java channel deadlocks cannot easily be ruled out completely. This is because the Java language specification allows the VM to throw a VirtualMachineError exception at any time [JLS, Java SE 8 Ed., §11.1]. Furthermore, if an exception reaches the top of a thread, the thread terminates but the program does not. Therefore, if a thread that holds a debit (i.e. an obligation to send) receives a VirtualMachineError and this exception kills the thread, then any thread that waits on this debit will deadlock. The root problem is that Java does not properly propagate failures. A language extension that would address this issue was proposed by Jacobs and Piessens [9].

7 Related Work

The proof assistant Coq includes a pure functional programming language with higher-order functions. Coq checks that all functions terminate. However, Coq's type system prevents a function from being passed as an argument to itself. Our approach supports methods that call themselves through dynamic binding, and can prove their termination.

Koka [10] is a functional programming language with effect inference, including the divergence effect. However, the inference algorithm is limited: it rules out recursion through the heap, which our approach supports.

Dafny [11] is a programming language that supports verification of termination, with powerful metrics. However, Dafny does not support dynamic binding of method calls.

Rudich et al. [12] verify well-formedness of *pure method specifications*. These may involve dynamically bound pure method calls.

8 Conclusion

We propose an approach for the modular specification and verification of total correctness properties of single-threaded and multithreaded object-oriented programs involving dynamically bound method calls. As far as we know, it is the first such approach. We propose a specification style that does not constrain implementations unnecessarily. The style enables any module to acquire private locks. The approach supports compare-and-swap loops. We sketch an encoding of liveness properties.

We have implemented our approach in a verification tool and validated it on a handful of small but challenging example programs. Further experimentation is needed, however, to see if our approach conveniently handles all program patterns.

Acknowledgements This work was supported in part by EU project ADVENT and by project G.0058.13 of the Research Foundation - Flanders (FWO).

References

1. Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, 2001.
2. Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, 2001.
3. Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In *POPL*, 2005.
4. Keiko Nakata and Tarmo Uustalu. Trace-based coinductive operational semantics for While. In *TPHOLs*, 2009.
5. Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *POPL*, 2008.
6. K. Rustan M. Leino, Peter Müller, and Jan Smans. Deadlock-free channels and locks. In *ESOP*, 2010.
7. John Boyland. Checking interference with fractional permissions. In *SAS*, 2003.
8. Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. Permission accounting in separation logic. In *POPL*, 2005.
9. Bart Jacobs and Frank Piessens. Failboxes: Provably safe exception handling. In *ECOOP*, 2009.
10. Daan Leijen. Koka: Programming with row polymorphic effect types. In *Mathematically Structured Functional Programming*, 2014.
11. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, 2010.

12. Arsenii Rudich, Ádám Darvas, and Peter Müller. Checking well-formedness of pure-method specifications. In *FM*, 2008.
13. Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *POPL*, 2011.

A Concurrency proof rules

In this appendix, we show our translation of the approach of Leino et al. [6] into separation logic.

We extend our programming language with commands for forking and joining threads, creating locks and channels, acquiring and releasing locks, and sending and receiving on channels:

$$\begin{aligned}\tau &::= \cdots \mid \textbf{thread} \mid \textbf{lock} \mid \textbf{channel}[\tau] \\ c &::= \cdots \mid \textbf{fork}\ c \mid e.\textbf{join}() \mid \textbf{new lock}() \mid \textbf{new channel}[\tau]() \\ &\quad \mid e.\textbf{acquire}() \mid e.\textbf{release}() \mid e.\textbf{send}(e) \mid e.\textbf{receive}()\end{aligned}$$

We extend our assertion language with constructs for describing threads, locks, and channels:

$$\begin{aligned}P, Q ::= \cdots &\mid t.\mathsf{thread}[\tau](w, Q) \mid \ell.\mathsf{lock}(w, \pi, I) \mid \ell.\mathsf{locked}(w, \pi, I, t) \\ &\mid \chi.\mathsf{channel}[\tau](w, P) \mid \chi.\mathsf{credit}() \mid \chi.\mathsf{debit}() \mid t.\mathsf{obs}(W)\end{aligned}$$

where t, w, ℓ, π, χ and W range over (expressions denoting) thread identifiers, wait levels, lock identifiers, fractions (rational numbers between zero, exclusive, and one, inclusive), channel identifiers, and bags of wait levels, respectively. $t.\mathsf{thread}[\tau](w, Q)$ asserts that thread t has result type τ, wait level w, and post-condition Q, where Q is an assertion with a free variable result denoting the thread result. $\ell.\mathsf{lock}(w, \pi, I)$ asserts fractional ownership with fraction π of lock ℓ created at wait level w with invariant I. $\ell.\mathsf{locked}(w, \pi, I, t)$ additionally denotes that thread t currently holds the lock. $\chi.\mathsf{channel}[\tau](w, P)$ asserts shared ownership of channel χ with element type τ created at wait level w with element predicate P, which is an assertion with a free variable $\mathsf{element}$ denoting an element. Whenever the channel holds elements $\overline{v}$, it owns the resources described by $\bigodot_{v \in \overline{v}} P[v/\mathsf{element}]$. $\chi.\mathsf{credit}()$ asserts ownership of one credit (receive permission) for channel χ. $\chi.\mathsf{debit}()$ asserts ownership of one debit (send obligation) for channel χ. $t.\mathsf{obs}(W)$ asserts that bag W collects the wait levels of thread t's obligations.

These assertions satisfy the following laws:

$$\begin{aligned}\ell.\mathsf{lock}(w, \pi_1 + \pi_2, I) &\Leftrightarrow \ell.\mathsf{lock}(w, \pi_1, I) * \ell.\mathsf{lock}(w, \pi_2, I) \\ \ell.\mathsf{lock}(w, 1, I) &\Rightarrow I \\ \chi.\mathsf{channel}[\tau](w, P) &\Rightarrow \chi.\mathsf{channel}[\tau](w, P) * \chi.\mathsf{channel}[\tau](w, P) \\ t.\mathsf{obs}(W) * \chi.\mathsf{channel}[\tau](w, P) &\Leftrightarrow t.\mathsf{obs}(W \uplus \{\!\{w\}\!\}) * \chi.\mathsf{channel}[\tau](w, P) \\ &\quad * \chi.\mathsf{debit}() * \chi.\mathsf{credit}()\end{aligned}$$

In words: lock fractions can be split and merged; full lock ownership allows the owner to destroy the lock, obtaining direct ownership of the protected resource;

channel fractions can be split; channel debits can be created and destroyed, which implies creating/destroying an obligation. Notice that any thread can destroy a debit, not just the one that created it.

The proof rules are shown in Figure 20.

$$\frac{W' \subseteq W \qquad \vdash \{\mathbf{tid}.\mathsf{obs}(W' \uplus \{w\}) * P\}\ c\ \{\mathbf{tid}.\mathsf{obs}(\{w\}) * Q[\mathsf{res}/\mathsf{result}]\}}{\vdash \{\mathbf{tid}.\mathsf{obs}(W) * P\}\ \mathbf{fork}\ c\ \{\mathbf{tid}.\mathsf{obs}(W - W') * \mathsf{res}.\mathsf{thread}(w, Q)\}}$$

$$\frac{}{\begin{array}{l}\{\mathbf{tid}.\mathsf{obs}(W) * t.\mathsf{thread}[\tau](w, Q) \wedge w \prec W\} \\ \vdash\ t.\mathbf{join}() \\ \{\mathbf{tid}.\mathsf{obs}(W) * Q[\mathsf{res}/\mathsf{result}]\}\end{array}}$$

$$\frac{}{\vdash \{I\}\ \mathbf{new\ lock}()\ \{\mathsf{res}.\mathsf{lock}(w, 1, I)\}}$$

$$\frac{}{\begin{array}{l}\{\mathbf{tid}.\mathsf{obs}(W) * \ell.\mathsf{lock}(w, \pi, I) \wedge w \prec W\} \\ \vdash\ t.\mathbf{acquire}() \\ \{\mathbf{tid}.\mathsf{obs}(W \uplus \{w\}) * \ell.\mathsf{locked}(w, \pi, I, \mathbf{tid}) * I\}\end{array}}$$

$$\frac{}{\begin{array}{l}\{\mathbf{tid}.\mathsf{obs}(W) * \ell.\mathsf{locked}(w, \pi, I, \mathbf{tid}) * I \wedge w \in W\} \\ \vdash\ t.\mathbf{release}() \\ \{\mathbf{tid}.\mathsf{obs}(W - \{w\}) * \ell.\mathsf{lock}(w, \pi, I)\}\end{array}}$$

$$\frac{}{\vdash \{\mathsf{true}\}\ \mathbf{new\ channel}[\tau]()\ \{\mathsf{res}.\mathsf{channel}[\tau](w, P)\}}$$

$$\frac{}{\vdash \{\chi.\mathsf{channel}[\tau](w, P) * P[v/\mathsf{element}]\}\ \chi.\mathbf{send}(v)\ \{\chi.\mathsf{credit}()\}}$$

$$\frac{}{\begin{array}{l}\{\mathbf{tid}.\mathsf{obs}(W) * \chi.\mathsf{channel}[\tau](w, P) * \chi.\mathsf{credit}() \wedge w \prec W\} \\ \vdash\ \chi.\mathbf{receive}() \\ \{\mathbf{tid}.\mathsf{obs}(W) * P[\mathsf{res}/\mathsf{element}]\}\end{array}}$$

Fig. 20. Separation logic proof rules for the approach of Leino et al. [6]. $w \prec W$ means $\forall w' \in W.\ w < w'$.

When a parent thread forks a child thread, the parent can pass some of its obligations to the child. The child is additionally charged with an obligation w, representing the obligation to terminate. w is the wait level of the new thread; it can be picked freely. The child must dispose of all of the obligations it received from its parent before it terminates.

Joining implies waiting, so the wait level of the target thread must be below the joining thread's obligations.

Creating a lock consumes the lock invariant. A wait level w for the lock can be picked freely.

When a thread acquires a lock, its thread identifier is recorded as an argument of the locked assertion. This ensures that the same thread releases the lock, as required by most lock implementations.

Releasing a lock requires that the lock's wait level is among the current thread's obligations. This could be invalidated by destroying a debit.

Sending on a channel produces a credit, and receiving consumes it.

B Soundness for concurrent programs

Here, we sketch an approach for proving soundness of a verification approach for total correctness properties of concurrent programs based on the ideas proposed in this paper.

We start from the verification approach for partial correctness properties of fine-grained concurrent programs of Jacobs and Piessens [13], and its soundness proof. In this proof, program semantics is described as a small-step relation $\rightsquigarrow$ on machine configurations γ. The goal is to prove that the special failure configuration **abort** is not reachable. The soundness proof defines a notion of a *valid configuration* $\mathsf{valid}(\gamma)$ and proves that if the program's correctness is provable by the proof rules of the approach, then the initial configuration is valid, **abort** is not valid, and validity is preserved by execution steps. It follows that each reachable configuration is valid and that **abort** is not reachable.

To formally state total correctness, we define the notion of a configuration γ *producing a behavior* O, denoted $\gamma \Downarrow O$, where O is either a configuration γ', denoting a finite run ending in a configuration γ', or $\bot$, denoting an infinite run. We define this judgment coinductively using the following inference rules:

$$\frac{\gamma \not\rightsquigarrow}{\gamma \Downarrow \gamma} \qquad\qquad \frac{\gamma \rightsquigarrow \gamma' \qquad \gamma' \Downarrow O}{\gamma \Downarrow O}$$

We say a behavior O *terminates* if O is a configuration where all threads have terminated. We say a program *terminates* if its initial configuration γ_0 only produces terminating behaviors:

$$\mathsf{program\_terminates} \Leftrightarrow \forall O.\ \gamma_0 \Downarrow O \Rightarrow O\ \mathsf{terminates}$$

We prove this by introducing the notion of a *valid configuration under a bag of ordinals* Λ, where Λ represents the total stock of call permissions available at that point. If a program's correctness is provable by the proof rules of our approach, then there exists a Λ_0 such that $\mathsf{valid}_{\Lambda_0}(\gamma_0)$. Furthermore, if $\gamma \rightsquigarrow \gamma'$ and $\mathsf{valid}_{\Lambda}(\gamma)$, then there exists a Λ' such that $\mathsf{valid}_{\Lambda'}(\gamma')$ and $(|\gamma'|, \Lambda') < (|\gamma|, \Lambda)$, where $|\gamma|$ is the syntactic size of γ. Indeed, at each execution step either some thread's current command is reduced to one that is smaller, or some thread performs a method call and a call permission is consumed. Soundness then follows by the following lemma:

Lemma 3.

$$\mathsf{valid}_\Lambda(\gamma) \wedge \gamma \Downarrow O \Rightarrow O \text{ terminates}$$

Proof. By well-founded induction on $(|\gamma|, \Lambda)$. That O is not a deadlocked configuration follows from the soundness of the approach of Leino et al. [6]; see their soundness proof.

C Soundness of the liveness approach

We adapt the approach of the preceding section to the liveness setting. First, we introduce machine steps that perform I/O, denoted $\overset{\mathsf{io}}{\rightsquigarrow}$. We define $\gamma \overset{\cdot}{\rightsquigarrow} \gamma'$ as $\gamma \overset{\mathsf{io}}{\rightsquigarrow} \gamma' \vee \gamma \rightsquigarrow \gamma'$. We then define a set of *traces*, coinductively using the following grammar:

$$\tau ::= \mathsf{io}{::}\tau \mid \gamma \mid \bot$$

We define the traces of a configuration, coinductively as follows:

$$\textsc{Trace-Normal}\ \frac{\gamma \not\rightsquigarrow}{\gamma \Downarrow \gamma} \qquad \textsc{Trace-Silent}\ \frac{\gamma \rightsquigarrow \gamma' \qquad \gamma' \Downarrow \tau}{\gamma \Downarrow \tau} \qquad \textsc{Trace-IO}\ \frac{\gamma \overset{\mathsf{io}}{\rightsquigarrow} \gamma' \qquad \gamma' \Downarrow \tau}{\gamma \Downarrow \mathsf{io}{::}\tau}$$

We define liveness of a trace, coinductively:

$$\textsc{Live-Term}\ \frac{\gamma\,\mathsf{terminated}}{\gamma\,\mathsf{live}} \qquad \textsc{Live-IO}\ \frac{\tau\,\mathsf{live}}{\mathsf{io}{::}\tau\,\mathsf{live}}$$

Here, $\gamma\,\mathsf{terminated}$ asserts that in γ all threads have terminated normally, i.e., no deadlock and no failure. The non-live traces are the ones that end in a failure or deadlocked configuration, or that eventually perform non-I/O steps forever.

We introduce a notion of *validity of a configuration under a bag of ordinals Λ and an I/O count N* , denoted $\mathsf{valid}_{\Lambda,N}(\gamma)$, such that

1. if a program is provably correct, then for all N_0, there exists a Λ_0 such that $\mathsf{valid}_{\Lambda_0,N_0}(\gamma_0)$ holds, where γ_0 is the program's initial configuration;
2. if $\gamma \rightsquigarrow \gamma'$ and $\mathsf{valid}_{\Lambda,N}(\gamma)$ then $\exists \Lambda'.\ \mathsf{valid}_{\Lambda',N}(\gamma')$ and $(|\gamma'|, \Lambda') < (|\gamma|, \Lambda)$; and
3. if $\gamma \overset{\mathsf{io}}{\rightsquigarrow} \gamma'$ and $\mathsf{valid}_{\Lambda,N}(\gamma)$ and $0 < N$, then $\exists \Lambda'.\ \mathsf{valid}_{\Lambda',N-1}(\gamma')$ and $(|\gamma'|, \Lambda') < (|\gamma|, \Lambda)$.
4. if $\mathsf{valid}_{\Lambda,N}(\gamma)$ then γ is not deadlocked and not failed.

We have the following corollary:

Lemma 4. *If $\gamma \overset{\cdot}{\rightsquigarrow} \gamma'$ and $\forall N.\ \exists \Lambda.\ \mathsf{valid}_{\Lambda,N}(\gamma)$, then $\forall N'.\ \exists \Lambda'.\ \mathsf{valid}_{\Lambda',N'}(\gamma')$.*

Proof. Fix a γ, γ', and N'. If $\gamma \rightsquigarrow \gamma'$, take $N = N'$; the goal follows easily. If $\gamma \overset{\mathsf{io}}{\rightsquigarrow} \gamma'$, take $N = N' + 1$; the goal follows easily.

Soundness then follows from the following lemma:

Lemma 5.

$$\forall \tau, \Lambda, \gamma.\ (\forall N.\ \exists \Lambda'.\ \mathsf{valid}_{\Lambda', N}(\gamma)) \wedge \mathsf{valid}_{\Lambda, 0}(\gamma) \wedge \gamma \Downarrow \tau \Rightarrow \tau\, \mathsf{live}$$

Proof. By coinduction. Fix τ. By well-founded induction on $(|\gamma|, \Lambda)$. By case analysis on $\gamma \Downarrow \tau$.

- **Case** Trace-Normal. By Live-Term.
- **Case** Trace-Silent. By the inner induction hypothesis.
- **Case** Trace-IO. By Live-IO and the outer induction hypothesis. (Take $N = 0$.)

□